<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Arrive home</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Buy Milk</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Leave forgrocery</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Look in fridge, no milk</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Arrive home</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example: Too Much Milk

Communicate and get a consistent view of the world (computer state)

What kind of knowledge and mechanisms do we need to get independent processes to

**Today: Synchronization**

**Review Questions:**

- Advantages? Disadvantages?
- How does each work?

**Scheduling Algorithms:**

- Lottery Scheduling
- Multilevel Feedback Queues
- SJF
- Round Robin
- FCFS

Last Class: CPU Scheduling
Thread A

- Do not buy any milk if there is no note (wait)
- Remove note (a version of unlock)
- Leave a note (a version of lock)

Restrict ourselves to atomic loads and stores as building blocks.

- Someone buys milk if you need it.
- Only one person buys milk at a time.

What are the correctness properties for this problem?

Too Much Milk: Solution 1

All synchronization involves waiting.

- Wait if locked
- Unlock when leaving a critical section or when access to shared data is complete
- Lock before entering a critical section, or before accessing shared data.

Lock: mechanism to prevent another process from doing something

Critical Section: piece of code that only one thread can execute at a time

Mutual Exclusion: ensure that only one thread does a particular activity at a time and excludes other threads from doing it at that time

Synchronization: use of atomic operations to ensure cooperation between threads

Synchronization Terminology
Does this work?

```c
remove note A:
{
    { 
        if (note B)
            } 
        if (note A)
            }
    leave note B
}

Thread A

Thread B
```

Too Much Milk: Solution 2

How about using labeled notes so we can leave a note before checking the

```c
remove note:
{
    { 
        if (note B)
            } 
        if (note A)
            }
    leave note B
}

Thread A

Thread B
```

Too Much Milk: Solution 3
This solution relies on loads and stores being atomic.

1. It is too complicated - it was hard to convince ourselves this solution

2. It is asymmetrical - thread A and B are different. Thus, adding more

modifications to existing threads.

3. A is busy waiting - A is consuming CPU resources despite the fact that it

is not doing any useful work.

\[ \text{Is Solution 3 a good solution?} \]

and then if B did not buy, it buys the milk.

removes note B. Since thread A loops, it waits for B to buy milk or not.

Thus, thread B buys milk (which thread A finds) or not, but either way it

that B bought or buys if it needed.

2. If there is no note, A waits until there is no longer a note B, and either finds milk

1. If there is not a note B, it is safe for A to buy since B has either not started or quit.

At point X, either there is a note B or not.

\[ \text{Correctness of Solution 3} \]
Lock is initially free.

Always release the lock after finishing with shared data.

Always acquire the lock before accessing shared data.

Rules for using a lock:

- **Lock**: acquire - wait until lock is free, then grab it.
- **Lock**: release - unlock, and wake up any thread waiting in acquire.

- **Routine**: provide mutual exclusion to shared data with two "atomic"

```
Locks
```

All of these require some hardware support, and waiting.

- **Monitors**: connects shared data to synchronization primitives.
- **Semaphores**: more general version of locks.

```
Locks: one process holds a lock at a time, does its critical section
```

- **Synchronization**: have your programming language provide atomic routines for

```
Language Support for Synchronization
```
<table>
<thead>
<tr>
<th>Concurrent Programs</th>
<th>高-级原语操作 (软件)</th>
<th>高-级原语操作 (硬件)</th>
<th>低-级原语操作</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>send &amp; receive</td>
<td></td>
<td>load/store</td>
</tr>
<tr>
<td></td>
<td>monikers</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>semaphores</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>locks</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

What we have and what we want

- Implementing high-level primitives requires low-level hardware support
- Hardware support for synchronization

---

How do we make Lock::Acquire and Lock::Release atomic?

```
Thread.A:
lock->Acquire();
{
    buy milk;
    if (milk)
      lock->Acquire();
    lock->Release();
}
Thread.B:
lock->Release();
```
Implementing Locks by Disabling Interrupts

System calls

Why not have the OS support lock::acquire() and lock::release as a system call?

Section

Delay handling any external events until after the thread is suspended with the critical section.

- External Events: Prevent these by disabling interrupts (e.g. call _disable_interrupts() in the kernel).
- Internal Events: Prevent these by not requesting any I/O operations during a critical section.

On multiprocessors, we can prevent the scheduler from getting control as follows:

Away from the running thread.

- External Events: The thread does something in the other processor
- Internal Events: The thread does something in its own processor

There are two ways the CPU scheduler gets control:
We will have a problem with multiprocessors.

enable interrupts every time it selects a new process to run.

disable interrupts when it switches to another thread. (The dispatcher can
already be on the ready queue. When the thread wakes up, it will go to
the ready queue. So, Release could put the thread on the ready queue but it
could also check the queue and not wake up the thread.

When should Acquire re-enable interrupts when going to sleep?

Example
Synchronization primitives are required to ensure that only one thread executes in a critical section at a time.

- Solution: use high-level primitives such as locks, semaphores, monitors
- Achieving synchronization directly with loads and stores is tricky and error-prone

- Critical sections identify pieces of code that cannot be executed in parallel
- Communication among threads is typically done through shared variables